Git\_版本控制

<http://blog.jobbole.com/25775/>

**1.1 关于版本控制**

什么是版本控制？我真的需要吗？

**版本控制是一种记录若干文件内容变化，以便将来查阅特定版本修订情况的系统。**

在本书所展示的例子中，我们仅对保存着软件源代码的文本文件作版本控制管理，但实际上，你**可以对任何类型的文件进行版本控制**。

如果你是位图形或网页设计师，可能会需要保存某一幅图片或页面布局文件的所有修订版本（这或许是你非常渴望拥有的功能）。采用**版本控制系统 （VCS）**是个明智的选择。有了它你就可以将某个文件回溯到之前的状态，甚至将整个项目都回退到过去某个时间点的状态。你可以比较文件的变化细节，查出最 后是谁修改了哪个地方，从而导致出现怪异问题，又是谁在何时报告了某个功能缺陷等等。使用版本控制系统通常还意味着，就算你乱来一气把整个项目中的文件改 的改删的删，你也照样可以轻松恢复到原先的样子。但额外增加的工作量却微乎其微。

**本地版本控制系统**

许多人习惯用复制整个项目目录的方式来保存不同的版本，或许还会改名加上备份时间以示区别。这么做唯一的好处就是简单。不过坏处也不少：有时候会混淆所在的工作目录，一旦弄错文件丢了数据就没法撤销恢复。

为了解决这个问题，人们很久以前就开发了许多种**本地版本控制系统**，大多都是采用某种简单的数据库来记录文件的历次更新差异（见图 1-1）。

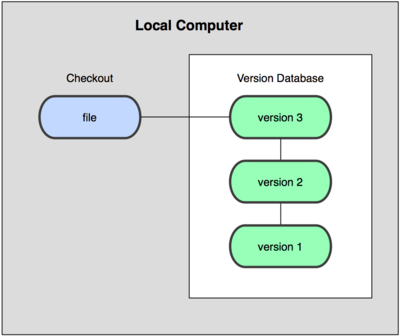


图 1-1. 本地版本控制系统

其中最流行的一种叫做 **rcs**，现今许多计算机系统上都还看得到它的踪影。甚至在流行的 Mac OS X 系统上安装了开发者工具包之后，也可以使用 rcs 命令。它的工作原理基本上就是**保存并管理文件补丁（patch）**。文件补丁是一种特定格式的文本文件，记录着对应文件修订前后的内容变化。所以，根据每次 修订后的补丁，rcs 可以通过不断打补丁，计算出各个版本的文件内容。

**集中化的版本控制系统**

接下来人们又遇到一个问题，如何让在不同系统上的开发者协同工作？于是，集中化的版本控制系统（ Centralized Version Control Systems，简称 CVCS ）应运而生。这类系统，诸如 CVS，Subversion 以及 Perforce 等，都有一个单一的集中管理的服务器，保存所有文件的修订版本，而协同工作的人们都通过客户端连到这台服务器，取出最新的文件或者提交更新。多年以来，这 已成为版本控制系统的标准做法（见图 1-2）。

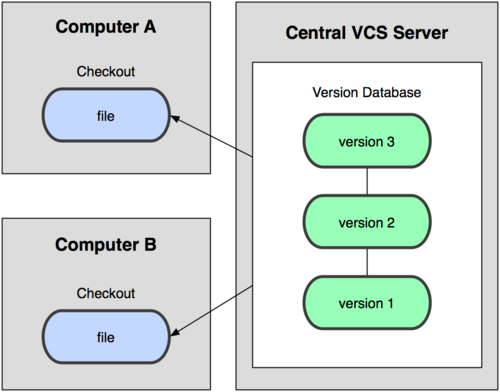


图 1-2. 集中化的版本控制系统

这种做法带来了许多好处，特别是相较于老式的本地 VCS 来说。现在，每个人都可以在一定程度上看到项目中的其他人正在做些什么。而管理员也可以轻松掌控每个开发者的权限，并且管理一个 CVCS 要远比在各个客户端上维护本地数据库来得轻松容易。

事分两面，有好有坏。这么做最显而易见的缺点是中央服务器的单点故障。如果宕机一小时，那么在这一小时内，谁都无法提交更新，也就无法协同工作。要 是中央服务器的磁盘发生故障，碰巧没做备份，或者备份不够及时，就还是会有丢失数据的风险。最坏的情况是彻底丢失整个项目的所有历史更改记录，而被客户端 提取出来的某些快照数据除外，但这样的话依然是个问题，你不能保证所有的数据都已经有人事先完整提取出来过。本地版本控制系统也存在类似问题，只要整个项 目的历史记录被保存在单一位置，就有丢失所有历史更新记录的风险。

**分布式版本控制系统**

于是**分布式版本控制系统（ Distributed Version Control System，简称 DVCS ）**面世了。在这类系统中，像 Git，Mercurial，Bazaar 以及 Darcs 等，客户端并不只提取最新版本的文件快照，而是把原始的代码仓库完整地镜像下来。这么一来，任何一处协同工作用的服务器发生故障，事后都可以用任何一个镜 像出来的本地仓库恢复。因为每一次的提取操作，实际上都是一次对代码仓库的完整备份（见图 1-3）。

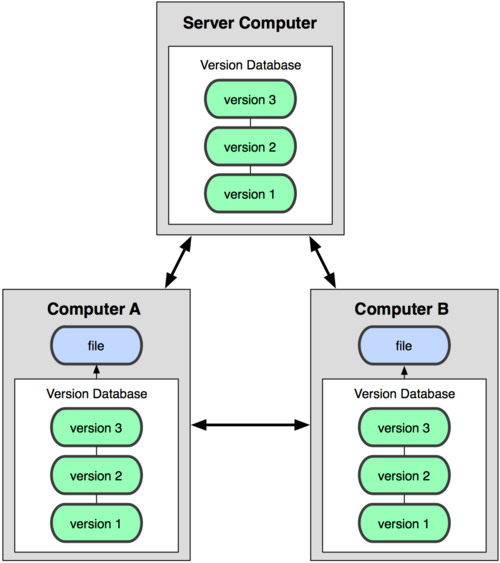


图 1-3. 分布式版本控制系统

更进一步，许多这类系统都可以指定和若干不同的远端代码仓库进行交互。籍此，你就可以在同一个项目中，分别和不同工作小组的人相互协作。你可以根据需要设定不同的协作流程，比如层次模型式的工作流，而这在以前的集中式系统中是无法实现的。

**1.2 Git 简史**

同生活中的许多伟大事件一样，Git 诞生于一个极富纷争大举创新的年代。Linux 内核开源项目有着为数众广的参与者。绝大多数的 Linux 内核维护工作都花在了提交补丁和保存归档的繁琐事务上（1991－2002年间）。到 2002 年，整个项目组开始启用分布式版本控制系统 BitKeeper 来管理和维护代码。

到了 2005 年，开发 BitKeeper 的商业公司同 Linux 内核开源社区的合作关系结束，他们收回了免费使用 BitKeeper 的权力。这就迫使 Linux 开源社区（特别是 Linux 的缔造者 Linus Torvalds ）不得不吸取教训，只有开发一套属于自己的版本控制系统才不至于重蹈覆辙。他们对新的系统制订了若干目标：

\* 速度 \* 简单的设计 \* 对非线性开发模式的强力支持（允许上千个并行开发的分支） \* 完全分布式 \* 有能力高效管理类似 Linux 内核一样的超大规模项目（速度和数据量）

自诞生于 2005 年以来，Git 日臻成熟完善，在高度易用的同时，仍然保留着初期设定的目标。它的速度飞快，极其适合管理大项目，它还有着令人难以置信的非线性分支管理系统（见第三章），可以应付各种复杂的项目开发需求。

**1.3 Git 基础**

那么，简单地说，Git 究竟是怎样的一个系统呢？请注意，接下来的内容非常重要，若是理解了 Git 的思想和基本工作原理，用起来就会知其所以然，游刃有余。在开始学习 Git 的时候，请不要尝试把各种概念和其他版本控制系统（诸如 Subversion 和 Perforce 等）相比拟，否则容易混淆每个操作的实际意义。Git 在保存和处理各种信息的时候，虽然操作起来的命令形式非常相近，但它与其他版本控制系统的做法颇为不同。理解这些差异将有助于你准确地使用 Git 提供的各种工具。

**直接记录快照，而非差异比较**

Git 和其他版本控制系统的主要差别在于，**Git 只关心文件数据的整体是否发生变化，而大多数其他系统则只关心文件内容的具体差异。**这类系统 （CVS，Subversion，Perforce，Bazaar 等等）每次记录有哪些文件作了更新，以及都更新了哪些行的什么内容，请看图 1-4。

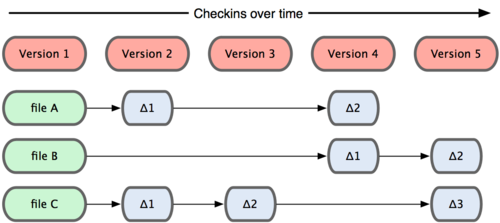


图 1-4. 其他系统在每个版本中记录着各个文件的具体差异

Git 并不保存这些前后变化的差异数据。实际上，Git 更像是把变化的文件作快照后，记录在一个微型的文件系统中。每次提交更新时，它会纵览一遍所有文件的指纹信息并对文件作一快照，然后保存一个指向这次快照 的索引。为提高性能，若文件没有变化，Git 不会再次保存，而只对上次保存的快照作一链接。Git 的工作方式就像图 1-5 所示。

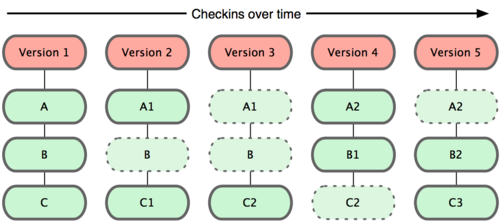


图 1-5. Git 保存每次更新时的文件快照

这是 Git 同其他系统的重要区别。它完全颠覆了传统版本控制的套路，并对各个环节的实现方式作了新的设计。Git 更像是个小型的文件系统，但它同时还提供了许多以此为基础的超强工具，而不只是一个简单的 VCS。稍后在第三章讨论 Git 分支管理的时候，我们会再看看这样的设计究竟会带来哪些好处。

**近乎所有操作都是本地执行**

在 Git 中的绝大多数操作**都只需要访问本地文件和资源，不用连网**。但如果用 CVCS 的话，差不多所有操作都需要连接网络。因为 Git 在本地磁盘上就保存着所有当前项目的历史更新，所以处理起来速度飞快。

举个例子，如果要浏览项目的历史更新摘要，Git 不用跑到外面的服务器上去取数据回来，而直接从本地数据库读取后展示给你看。所以任何时候你都可以马上翻阅，无需等待。如果想要看当前版本的文件和一个月 前的版本之间有何差异，Git 会取出一个月前的快照和当前文件作一次差异运算，而不用请求远程服务器来做这件事，或是把老版本的文件拉到本地来作比较。

用 CVCS 的话，没有网络或者断开 VPN 你就无法做任何事情。但用 Git 的话，就算你在飞机或者火车上，都可以非常愉快地频繁提交更新，等到了有网络的时候再上传到远程仓库。同样，在回家的路上，不用连接 VPN 你也可以继续工作。换作其他版本控制系统，这么做几乎不可能，抑或非常麻烦。比如 Perforce，如果不连到服务器，几乎什么都做不了（译注：默认无法发出命令p4 edit file 开始编辑文件，因为 Perforce 需要联网通知系统声明该文件正在被谁修订。但实际上手工修改文件权限可以绕过这个限制，只是完成后还是无法提交更新。）；如果是 Subversion 或 CVS，虽然可以编辑文件，但无法提交更新，因为数据库在网络上。看上去好像这些都不是什么大问题，但实际体验过之后，你就会惊喜地发现，这其实是会带来很大不同的。

**时刻保持数据完整性**

**在保存到 Git 之前，所有数据都要进行内容的校验和（checksum）计算，并将此结果作为数据的唯一标识和索引。**换句话说，不可能在你修改了文件或目录之后，Git 一无所知。这项特性作为 Git 的设计哲学，建在整体架构的最底层。所以如果文件在传输时变得不完整，或者磁盘损坏导致文件数据缺失，Git 都能立即察觉。

Git 使用 SHA-1 算法计算数据的校验和，通过对文件的内容或目录的结构计算出一个 SHA-1 哈希值，作为指纹字符串。该字串由 40 个十六进制字符（0-9 及 a-f）组成，看起来就像是：

Shell

|  |  |
| --- | --- |
| 1 | 24b9da6552252987aa493b52f8696cd6d3b00373 |

Git 的工作完全依赖于这类指纹字串，所以你会经常看到这样的哈希值。实际上，所有保存在 Git 数据库中的东西都是用此哈希值来作索引的，而不是靠文件名。

**多数操作仅添加数据**

常用的 Git 操作大多仅仅是把数据添加到数据库。因为任何一种不可逆的操作，比如删除数据，都会使回退或重现历史版本变得困难重重。在别的 VCS 中，若还未提交更新，就有可能丢失或者混淆一些修改的内容，但在 Git 里，一旦提交快照之后就完全不用担心丢失数据，特别是养成定期推送到其他仓库的习惯的话。

这种高可靠性令我们的开发工作安心不少，尽管去做各种试验性的尝试好了，再怎样也不会弄丢数据。至于 Git 内部究竟是如何保存和恢复数据的，我们会在第九章讨论 Git 内部原理时再作详述。

**文件的三种状态**

好，现在请注意，接下来要讲的概念非常重要。**对于任何一个文件，在 Git 内都只有三种状态：已提交（committed），已修改（modified）和已暂存（staged）。**已提交表示该文件已经被安全地保存在本地数据库 中了；已修改表示修改了某个文件，但还没有提交保存；已暂存表示把已修改的文件放在下次提交时要保存的清单中。

由此我们看到 Git 管理项目时，**文件流转的三个工作区域：Git 的工作目录，暂存区域，以及本地仓库。**

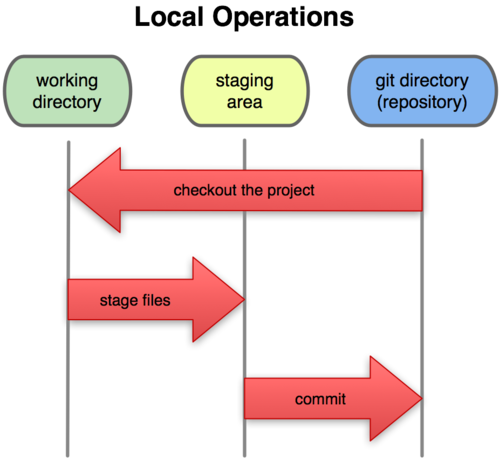


图 1-6. 工作目录，暂存区域，以及本地仓库

每个项目都有一个 Git 目录（译注：如果 git clone 出来的话，就是其中 .git 的目录；如果git clone --bare 的话，新建的目录本身就是 Git 目录。），它是 Git 用来保存元数据和对象数据库的地方。该目录非常重要，每次克隆镜像仓库的时候，实际拷贝的就是这个目录里面的数据。

从项目中取出某个版本的所有文件和目录，用以开始后续工作的叫做**工作目录**。这些文件实际上都是从 Git 目录中的压缩对象数据库中提取出来的，接下来就可以在工作目录中对这些文件进行编辑。

所谓的**暂存区域**只不过是个简单的文件，一般都放在 Git 目录中。有时候人们会把这个文件叫做索引文件，不过标准说法还是叫暂存区域。

**基本的 Git 工作流程如下：**

**1. 在工作目录中修改某些文件。 2. 对修改后的文件进行快照，然后保存到暂存区域。 3. 提交更新，将保存在暂存区域的文件快照永久转储到 Git 目录中。**

所以，我们可以从文件所处的位置来判断状态：如果是 Git 目录中保存着的特定版本文件，就属于已提交状态；如果作了修改并已放入暂存区域，就属于已暂存状态；如果自上次取出后，作了修改但还没有放到暂存区域，就 是已修改状态。到第二章的时候，我们会进一步了解其中细节，并学会如何根据文件状态实施后续操作，以及怎样跳过暂存直接提交。